Pyramid: A large-scale array-oriented active storage system

Viet-Trung Tran, Bogdan Nicolae, Gabriel Antoniu, Luc Bougé

Pyramid: A large-scale array-oriented active storage system

Viet-Trung Tran
ENS Cachan
Rennes, France
Viet-trung.Tran@irisa.fr

Bogdan Nicolae
INRIA Saclay
Orsay, France
Bogdan.Nicolae@inria.fr

Gabriel Antoniu
INRIA/IRISA
Rennes, France
Gabriel.Antoniu@inria.fr

Luc Bougé
ENS Cachan
Rennes, France
Luc.Bouge@bretagne.ens-cachan.fr

ABSTRACT

The recent explosion in data sizes manipulated by distributed scientific applications has prompted the need to develop specialized storage systems capable to deal with specific access patterns in a scalable fashion. In this context, a large class of applications focuses on parallel array processing: small parts of huge multi-dimensional arrays are concurrently accessed by a large number of clients, both for reading and writing. A specialized storage system that deals with such an access pattern faces several challenges at the level of data/metadata management. We introduce Pyramid, an active array-oriented storage system that addresses these challenges and shows promising results in our initial evaluation.

1. INTRODUCTION

Scientific applications are increasingly data-intensive: a variety of disciplines including astronomy, biology, physics, oceanography, atmospheric sciences, and climatology generate and process huge amounts of data that have reached the exa-scale. In this context, the scalability of data management becomes a critical issue, as it affects the performance of the whole application.

Many established storage solutions such as parallel file systems and database management systems strive to achieve high-performance and scalability of data accesses under concurrency. However, they expose data access models (e.g. file systems, structured databases) that are too general and do not exactly match the natural requirements of the application. This forces the application developer to either adapt to the exposed data access model or to use an intermediate layer that performs a translation. In either case, the mismatch leads to suboptimal data management: as noted in [9], the one-storage-solution-fits-all-needs has reached its limits.

Thus, there is an increasing need to specialize the I/O stack to match the requirements of the application. In scientific computing, of particular interest is a large class of applications that represent and manipulate data as huge multi-dimensional arrays [6]. Such applications typically consist of a large number of distributed workers that concurrently process subdomains of those arrays. Besides the ability to sustain a high throughput for parallel subdomain processing, an important requirement in this context is versioning: according to [8], scientific applications often need to access past snapshots of the data (e.g. in order to visualize a phenomenon as it progresses in time).

In this paper we address both requirements mentioned above. We propose *Pyramid*, a specialized array-oriented storage manager, specifically optimized for parallel array processing.

Relationship to authors' previous work. Pyramid is inspired by BlobSeer [4, 10], a versioning-oriented storage system specifically optimized to sustain a high throughput under concurrency. Where BlobSeer focuses on unstructured Binary Large OBjects (BLOBs) that represent linear addressing spaces, Pyramid generalizes the same principles for multi-dimensional data and introduces several new features.

New contributions. Our contributions can be summarized as follows: (1) we introduce a dedicated array-oriented data access model that offers support for active storage and versioning; (2) we enrich striping techniques specifically optimized for multi-dimensional arrays [5] with a distributed metadata management scheme that avoids potential I/O bottlenecks observed with centralized approaches; (3) we evaluate Pyramid for highly concurrent data access patterns and report preliminary results.

2. RELATED WORK

SciDB [3, 8] is a recent effort to design a science-oriented database system from scratch. SciDB departs from the relational database model to offer an array-oriented database model. It aims at providing chunking layout, versioning

data, however, these features are only briefly mentioned for the importance. A concrete solution to address them is still on-going work. Moreover, SciDB does not address chunking layout management which may be a problem in exa-scale. In this paper, we present an approach which aims at providing these features.

Emad et al. introduced ArrayStore [7], a storage manager for complex, parallel array processing. Similarly to SciDB, ArrayStore partitions large arrays into chunks and stores them in a distributed fashion. ArrayStore organizes metadata as R-trees, which are maintained in a centralized fashion. This can represent a potential bottleneck for the limit on the number of concurrent processes that can be served simultaneously. Furthermore, ArrayStore is only designed as a read-only storage system. The authors acknowledge that ArrayStore may be poorly handling write workloads due to its two chunking levels. Our system is designed for both read and write workloads.

3. GENERAL DESIGN

Array versioning. At the core of our approach is the idea of representing data updates using immutable data and metadata. Whenever a multi-dimensional array needs to be updated, the affected cells are never overwritten, but rather a new snapshot of the whole array is created, into which the update is applied. In order to offer the illusion of fully-independent arrays with minimal overhead in both storage space utilization and performance, we rely on differential updates: only the modified cells are stored for each new snapshot. Any unmodified data or metadata is shared between the new snapshot and the previous snapshots. The metadata of the new snapshot is generated in such way that it seamlessly interleaves with the metadata of the previous snapshots to create incremental snapshots that look and act at application level as independent arrays.

Active storage support. Many datacenters nowadays are made of machines equipped with commodity hardware that often act as both storage elements and compute elements. In this context, it is highly desirable to be able to move the computation to the data rather than the other way around, for two reasons: (1) it conserves bandwidth, which is especially important when data transfers are expensive (e.g. because of cost concerns on clouds or because of high latency / low bandwidth); (2) it enables better workload parallelization, as part of the work can be delegated to the storage elements (e.g. post-processing filters, compression, etc.).

Versioning array-oriented access interface. We propose an interface to multi-dimensional data that is specifically designed to enable fine-grained versioning access to subdomains while offering the features mentioned above.

id = CREATE(n, sizes[], defval)

creates a n-dimensional array identified by id and spanning sizes[i] cells in each dimension $0 \leq i < n$. By convention, the initial snapshot associated to the array has version number 0 and all cells are filled with the default initial value defval. This is a lazy initialization: no data and metadata is added until some cells of the array are actually updated.

READ(id, v, offsets[], sizes[], buffer)

reads a subdomain from the snapshot v of the array id. The subdomain is delimited by offsets[i] and spans sizes[i] cells in each dimension $0 \leq i < n$. The contents of the subdomain is stored in the local memory region buffer.

w = WRITE(id, offsets[], sizes[], buffer)

writes the content of the local memory region buffer to the cells of the subdomain delimited by offsets[i] and sizes[i] in each dimension $0 \leq i < n$ of the array id. The result is a new snapshot whose version number is w.

w = SEND_COMPUTATION(id, v, offsets[], sizes[], f)

applies function f to all cells of the subdomain delimited by offsets[i] and sizes[i] in each dimension $0 \leq i < n$. The result is a new snapshot whose version number is w. The computation is performed remotely on the storage elements and involves no additional data transfers.

Multi-dimensional aware chunking. Chunking is a standard approach to reduce contention for parallel accesses to multi-dimensional data [5]. The core idea is very simple: split the array into chunks and distribute them among the storage elements, which results in a distribution of the I/O workload.

In this context, the partitioning scheme plays a crucial role: under unfavorable conditions, read and write queries may generate "strided" access patterns (i.e. access small parts from a large number of chunks), which has a negative impact on performance. To reduce this effect, we propose to split the array into subdomains that are equally sized in each dimension. Using this approach, the neighbors of cells have a higher chance of residing in the same chunk irrespective of the query type, which greatly limits the number of chunks that need to be accessed.

Furthermore, this scheme brings an important advantage for active storage: since data is distributed among multiple storage elements, so does any computation that is sent to the data, leading to an efficient implicit parallelization.

Lock-free, distributed chunk indexing. Data is striped and stored in a distributed fashion, which implies that additional metadata is necessary to describe the composition of arrays in terms of chunks and where these chunks can be found.

The problem of building spatial indexes has been studied extensively, with several specialized data structures proposed: R-trees, xd-trees, quad-trees, etc. Most of these structures were originally designed and later optimized for centralized management.

However, a centralized metadata management scheme limits its scalability as in distributed file systems. Even in the situation when enough storage is available for storage of metadata, the metadata server can quickly become a bottleneck when attempting to serve a large number of clients simultaneously.

Thus, it is important to implement a distributed metadata management scheme. To this end, we propose a distributed quad-tree like structure that is used to index the chunk layout and is specifically optimized for concurrent updates. Our scheme takes advantage of the fact that data and metadata remains immutable in order to efficiently handle concurrent metadata updates without locking.

4. PYRAMID

4.1 Architecture

Pyramid is a distributed system consisting of the following components:

Version managers are the core of Pyramid. They coordinate the process of assigning new snapshot versions for concurrent writes such that total ordering is guaranteed. At the same time, they wait for the moment when snapshots are consistent and expose them to the readers in an atomic fashion.

Metadata managers implement the distributed quad-trees introduced in the previous section. They are responsible for instructing the clients what chunks to fetch from what location for any given subdomain.

Active storage servers physically store chunks generated by creating new arrays or updating existing arrays. An active storage server can also execute handler functions assigned to each object.

A storage manager is in charge of monitoring all available storage servers and schedule the placement of newly created chunks based on the monitoring information.

4.2 Zoom on chunk indexing

In Section 3 we argued in favor of a distributed chunk indexing scheme that leverages versioning to avoid potentially expensive locking under concurrency. In this section we briefly describe how to achieve this objective by introducing a distributed tree structure that is specifically designed to take advantage of the fact that data and metadata remains immutable.

Our solution generalizes the metadata management proposed in [4], which relies on the same principle to achieve high metadata scalability under concurrency. For simplicity, we illustrate our approach for a two-dimensional array in the rest of this section, a case that corresponds to a quad-tree (i.e. each inner node has four children). The same approach can be easily generalized for an arbitrary number of dimensions.

Structure of the distributed quad-tree. We make use of a partitioning scheme that recursively splits the initial two-dimensional array into four subdomains, each corresponding to one of the four quadrants: Upper-Left (UL), Upper-Right (UR), Bottom-Left (BL), Bottom-Right (BR). This process continues until a subdomain size is reached that is small enough to justify storing the entire subdomain as a single chunk. To each subdomain obtained in this fashion, we associate a tree node (said to "cover" the subdomain) as follows: the leaves cover single chunks (i.e. they hold information about what storage servers store the chunks), the inner nodes have four children and cover the subdomain formed by the quadrants (i.e. UL, UR, BL, BR), while the root covers the whole array.

All tree nodes are labeled with a version number (initially 0) that corresponds to the snapshot to which they belong. Updates to the array generate new snapshots with increasing version numbers. Inner nodes may have as children nodes that are labeled with a smaller version number, which effectively enables sharing of unmodified data and their whole corresponding sub-trees between snapshots. Figure 1 illustrates this for an initial version of the array to which an update is applied.

Since tree nodes are immutable, they are uniquely identified by the version number and the subdomain they cover. Based on this fact, we store the resulting tree nodes persistently in a distributed fashion, using a Distributed Hash Table (DHT) maintained by the metadata managers: for each tree node a corresponding key-value pair is generated and added. Thanks to the DHT, accesses to the quad-tree are distributed under concurrency, which eliminates metadata bottlenecks present in centralized approaches.

Read queries. A read query descends into the quad tree in a top-down fashion: starting from the root, it recursively visits all quadrants that cover the requested subdomain of the read query until all involved chunks are discovered. Once this step is completed, the chunks are fetched from the corresponding storage servers and brought locally. Note that the tree nodes remain immutable, which enables reads to proceed in parallel with writes, without the need to synchronize quad tree accesses.

Write queries. A write query first sends the chunks to the corresponding storage servers and then builds the quad-tree associated to the new snapshot of the array. This is a bottom-up process: first the new leaves are added in the DHT, followed by the inner nodes up to the root. For each inner node, the four children are established: some may belong to earlier snapshots. Under a concurrent write access pattern, this scheme apparently implies a synchronization of the quad-tree generation, because of inter-dependencies between tree nodes. However, we avoid such a synchronization by feeding additional information about the other concurrent writers during the quad-tree generation. This enables each writer to "predict" what tree nodes will be generated by the other writers and use those tree nodes as children if necessary, under the assumption that the missing children will be eventually added to the DHT by the other writers.

5. PRELIMINARY EVALUATION

We performed a set of preliminary experiments that aims to evaluate both the performance and the scalability of our approach under concurrent accesses. To this end, we simulated a common access pattern exhibited by scientific applications: 2D array dicing. This access pattern involves a large number of processes that read and write distinct parts of the same large array in parallel.

In this context, we measure the scalability of our approach: the aggregated throughput achieved for an increasing number of concurrent processes, when keeping the subdomain size corresponding to each process constant. More specifically, each process reads and writes a 1 GB large subdomain that consists of 32x32 chunks (i.e. 1024x1024 bytes for each chunk). We start with an array that holds a single such subdomain (i.e. it corresponds to a single process) and gradually increase its size to 2x2, 3x3, ... 7x7 subdomains (which corresponds to 4, 9, ... 49 parallel processes).

The processes are deployed on the nodes of the Graphene cluster, which consists of nodes equipped with x86_64 CPUs that are interconnected though an 1 Gbps Ethernet network. Each process is deployed on a dedicated node (for a total of 49 nodes), while the rest of the nodes are used to deploy Pyramid in the following configuration: 1 version manager, 1 storage manager, 30 metadata servers, and 48 object stor-

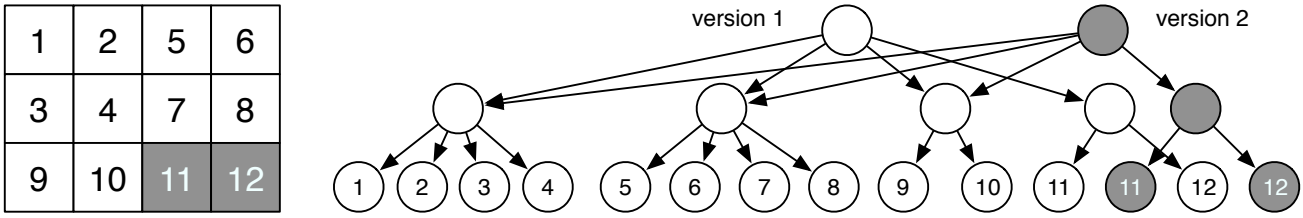

Figure 1: Metadata quad-trees by example: two chunks (dark color) of an initial array (partitioned according to the left figure) are updated, leading to the additional tree nodes and their links represented in the right figure.

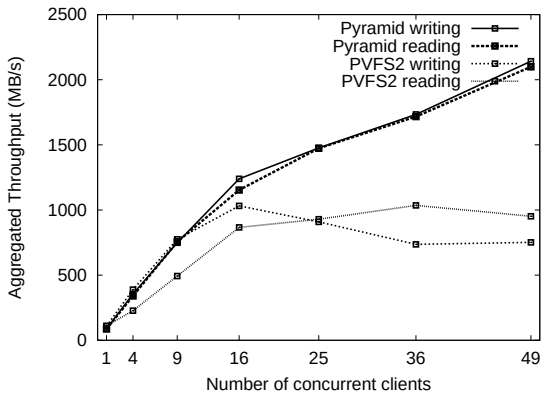

Figure 2: Aggregated throughput achieved under concurrency

age servers. We compare our approach to the case when a standard parallel file system is used to store the whole array as a single sequence of bytes in a file. To this end, we deployed an instance of PVFS2 [2] on the same 80 nodes used to evaluate Pyramid.

Results are shown in Figure 2. As can be observed, the throughput reaches 80 MB/s for one single client, demonstrating high performance even for fine granularity decompositions. Furthermore, with increasing number of concurrent clients, the aggregated throughput grows steadily up to 2.1 GB/s. This illustrates good scalability for our approach, which is a consequence of both the multi-dimensional aware data striping and the distributed metadata management. On the other hand, the scalability of PVFS2 suffers because the array is represented as a single sequence of bytes, which leads to fine-grain, strided access patterns.

6. CONCLUSIONS

We introduced Pyramid, an array-oriented active storage system that can efficiently address the I/O needs of parallel array processing. Preliminary evaluation shows promising results: our prototype demonstrated good performance and scalability under concurrency, both for read and write workloads.

In future work, we plan to explore the possibility of using Pyramid as a storage backend for SciDB [3] and HDF5 [1]. This direction has a high potential to improve I/O throughput while keeping compatibility with standardized data access interfaces. Another promising direction for our approach are scientific applications that process arrays at different resolutions: many times whole subdomains (e.g. zero-filled regions) can be characterized by simple summary information. In this context, our distributed metadata scheme can be enriched to hold such summary information about the subdomains in the tree nodes, which can be relied upon to avoid deeper fine-grain accesses when possible.

7. REFERENCES

[1] http://www.hdfgroup.org/hdf5/.

[2] http://www.pvfs.org/.

[3] P. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 963–968. ACM, 2010.

[4] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. BlobSeer: Next-generation data management for large scale infrastructures. *Journal of Parallel and Distributed Computing*, 71:169–184, February 2011.

[5] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Data Engineering, 1994. Proceedings.10th International Conference*, pages 328 –336, feb 1994.

[6] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O requirements of scientific applications: An evolutionary view. In *HPDC '02: Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*, pages 49–59. IEEE, 2002.

[7] E. Soroush, M. Balazinska, W. Daniel, and M. Pack. Arraystore: A storage manager for complex parallel array processing. In *SIGMOD '11: Proceedings of the SIGMOD Conference*, number 1, 2011.

[8] M. Stonebraker, J. Becla, D. Dewitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for Science Data Bases and SciDB. In *CIDR 2009: Conference on Innovative Data Systems Research*. Citeseer, 2009.

[9] M. Stonebraker and U. Cetintemel. One size fits all: An idea whose time has come and gone. In *ICDE 2005: Proceedings of the 21st International Conference on Data Engineering*, pages 2–11. IEEE, 2005.

[10] V.-T. Tran, B. Nicolae, G. Antoniu, and L. Bougé. Efficient support for MPI-IO atomicity based on versioning. In *11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011)*. IEEE CS Press, 2011.